KAPITEL 2

Den Programmablauf steuern

Das Steuern des Programmablaufs bedeutet, zu beeinflussen, wie Anweisungen in einer Anwendung ausgeführt werden. Es gibt klassische Ablaufsteuerungen, wie zum Beispiel bedingte Anweisungen und Schleifen, die die Reihenfolge der Anweisungen festlegen. Dart bietet eine Reihe von Methoden an, um den Ablauf der Anwendung abhängig von Entscheidungen zu steuern.

Haben Sie schon andere Sprachen wie Python, JavaScript und so weiter genutzt, werden Sie mit den Inhalten aus diesem Kapitel sehr vertraut sein. Wer neu in der Entwicklung ist, wird dieses Kapitel dringend benötigen! Anweisungen zum Steuern des Programmablaufs sind in den meisten Sprachen, mit denen man zu tun hat, sehr ähnlich. Ein Teil des Lernens einer Sprache dreht sich darum, sich mit dieser Art von Anweisungen vertraut zu machen.

In diesem Kapitel werden Sie lernen, wie Sie den Programmablauf steuern, um Logik in Ihre Anwendung zu bringen. Sie werden zudem Anwendungsfälle für jede Anweisung kennenlernen. Viele der Abläufe nutzen eine bedingte Anweisung, die dazu dient, vorzugeben, welche Aktionen auszuführen sind. Achten Sie besonders auf solche Bedingungen und versuchen Sie, den Programmablauf in Ihrer Anwendung effizient zu steuern.

Überprüfen, ob eine Bedingung erfüllt wurde

Problem

Sie wollen eine logische Prüfung für eine Bedingung durchführen, bevor Sie eine Anweisung ausführen.

Lösung

Nutzen Sie eine `if`-Anweisung, um eine Ablaufsteuerung für eine binäre Option zu erhalten. Mit einer `if`-Anweisung können Sie prüfen, ob eine logische Aussage gültig ist.

Gibt es mehrere Optionen, sollten Sie über den Einsatz einer `switch`-Anweisung nachdenken (siehe Abschnitt »Aktionen abhängig von einem Wert ausführen«).

Dieses Beispiel zeigt, wie Sie die `if`-Bedingung nutzen. Sie dient dazu, den Wert einer `bool`-Variablen zu prüfen. Ist diese auf `true` gesetzt, wird die erste Nachricht ausgegeben. Ist die `bool`-Variable auf `false` gesetzt, wird die Alternative ausgegeben.

```
void main() {

  bool isFootball = true;

  if (isFootball) {
    print('Football ist toll!');
  } else {
    print('Sport ist toll!');
  }
}
```

Diskussion

Mit einer `if`-Anweisung erlangen Sie die Kontrolle über den logischen Ablauf in einer Anwendung. Solch eine Steuerung ist für das Bauen von Anwendungen essenziell und Sie erhalten so einen einfachen Mechanismus, um bei Entscheidungen auswählen zu können. Eine bedingte `if`-Anweisung ist in Abbildung 2-1 zu sehen.

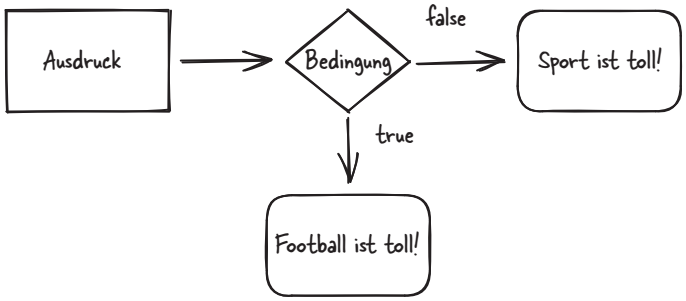

Abbildung 2-1: Ablauf einer `if`*-Anweisung*

In diesem Beispiel ist die Überprüfung bei der `if`-Anweisung implizit – sie prüft, ob der zugewiesene Wert `true` ist. Logische Operatoren wie && (AND) und || (OR) können ebenfalls genutzt werden, um den zu prüfenden Ausdruck zu erweitern. Mit dem AND-Operator kontrollieren Sie, ob beide Ausdrücke `true` sind, bevor der Code ausgeführt wird. Ein logisches OR dient dazu, sicherzustellen, dass einer oder mehrere der Ausdrücke `true` sind. Zudem können die logischen Operatoren durch den Einsatz des !-Operators (Invert) auf einen booleschen Wert umgekehrt werden.

Der typische Anwendungsfall für eine `if`-Anweisung ist das Treffen einer Entscheidung zwischen zwei oder mehr Optionen. Haben Sie nur zwei Optionen, ist diese Art von Ablaufsteuerung ideal.

Es gibt auch noch ein »Collection-if«, das zusätzliche Funktionalität zum Testen eines Elements mitbringt. Die mit dem Element verbundenen Daten lassen sich abhängig von Bedingungen im Collection-Objekt setzen – zum Beispiel, wenn es das erste oder letzte Objekt in einer Datenstruktur ist.

Iterieren, bis eine Bedingung erfüllt ist

Problem

Eine Methode soll laufen, bis eine Bedingung in einer Anwendung erfüllt ist.

Lösung

Nutzen Sie eine `while`-Schleife, wenn die Eintrittsbedingung zu Beginn des Programmablaufs überprüft werden soll. Die Prüfung findet also am Anfang der Schleife statt. Eine `while`-Schleife läuft also minimal gar nicht und maximal *N*-mal durch.

Dies ist ein Beispiel für eine Ablaufsteuerung durch eine `while`-Schleife:

```
void main() {

  bool isTrue = true;

  while (isTrue) {
    print ('Hallo');
    isTrue = false;
  }
}
```

Nutzen Sie eine `do while`-Schleife, wenn sie mindestens einmal ausgeführt werden soll.

Wie Sie in Abbildung 2-2 sehen, wird die Schleifenbedingung beim Eintritt in die Schleife überprüft – also vor jeder Iteration.

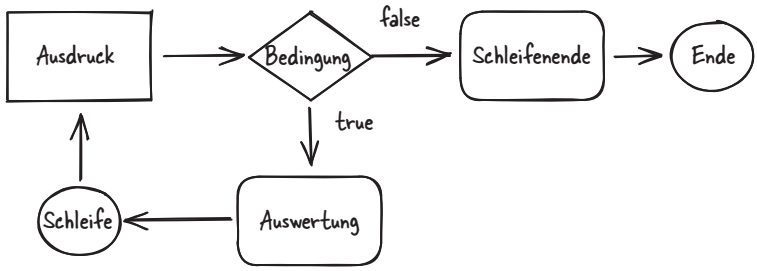

Abbildung 2-2: Logik einer `while`*-Schleife*

Hier ein Beispiel für eine Ablaufsteuerung mit einer `do while`-Schleife:

```
void main() {

  bool isTrue = true;

  do {
    print ('Hallo');
    isTrue = false;
  } while (isTrue) ;
}
```

Beachten Sie den Unterschied in Abbildung 2-3 zum vorigen Beispiel mit der `while`-Bedingung.

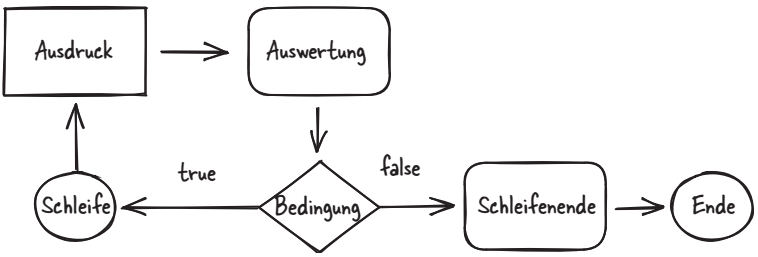

Abbildung 2-3: Logik einer `do while`*-Schleife*

Bei dieser Kontrollstruktur wird die Bedingung am Ende der Anweisungen ausgeführt – die Schleife läuft also mindestens einmal durch.

Diskussion

Der entscheidende Unterschied in diesen Beispielen ist die Art der Ausführung und was das für den Programmablauf bedeutet.

Im Beispiel mit der `while`-Schleife gibt die Demo-Anwendung nur einen Wert aus, wenn die `bool`-Variable auf `true` gesetzt ist. Das Beispiel mit der `do while`-Schleife führt auf jeden Fall eine `print`-Anweisung aus – unabhängig vom initialen Wert der Variablen `isTrue`.

Eine `while`-Schleife prüft eine Bedingung, bevor die Schleife ausgeführt wird; Sie können damit also 0 ... *N* Iterationen ablaufen lassen. Ein typischer Anwendungsfall besteht darin, eine Variable zu verwenden, um die Anzahl der Iterationen zu steuern.

Bei einer `do while`-Anweisung besteht der typische Anwendungsfall hingegen darin, die Schleife mindestens einmal laufen zu lassen. In solchen Situationen ist die Wahl einer `do while`-Schleife also besser.

Über eine Reihe von Elementen iterieren

Problem

Eine Methode soll einen definierten Bereich von Elementen durchlaufen.

Lösung

Nutzen Sie eine `for`-Anweisung, um eine definierte Anzahl von Iterationen in einem definierten Bereich auszuführen. Der angegebene Bereich wird als Teil der Initialisierung der `for`-Anweisung bestimmt.

Dies ist ein Beispiel für eine `for`-Anweisung:

```
void main() {

  int maxIterations = 10;
  for (var i = 0; i < maxIterations; i++) {
    print ('Iteration: $i');
  }
}
```

Haben Sie Objekte, die über eine Schleife erreichbar (also iterierbar) sind, können Sie auch `forEach` nutzen:

```
void main() {
  List daysOfWeek = ['Sonntag', 'Montag', 'Dienstag'];

  daysOfWeek.forEach((print));
}
```

Diskussion

Eine `for`-Anweisung lässt sich in einer Vielzahl von Fällen einsetzen, wie zum Beispiel beim Ausführen einer exakten Zahl von Aktionen (beispielsweise beim Initialisieren von Variablen).

In Abbildung 2-4 dient die `for`-Schleife dazu, über Elemente zu iterieren. Wie bei einer `while`-Schleife wird die Bedingung zu Beginn geprüft. Trifft sie zu, endet die Schleife. Wenn nicht, läuft die Schleife weiter, bis der Bereich der Elemente abgearbeitet ist.

Wie das zweite Beispiel zeigt, ist eine `forEach`-Anweisung eine sehr nützliche Technik, um auf Informationen in einem Objekt zuzugreifen. Haben Sie einen iterierbaren Typ (zum Beispiel das `List`-Objekt), bietet eine `forEach`-Anweisung die Möglichkeit, direkt auf den Inhalt zuzugreifen. Das Anfügen von `forEach` an das `List`-Objekt dient als Abkürzung, bei der eine `print`-Anweisung direkt auf jedes Element in der Liste angewendet werden kann.

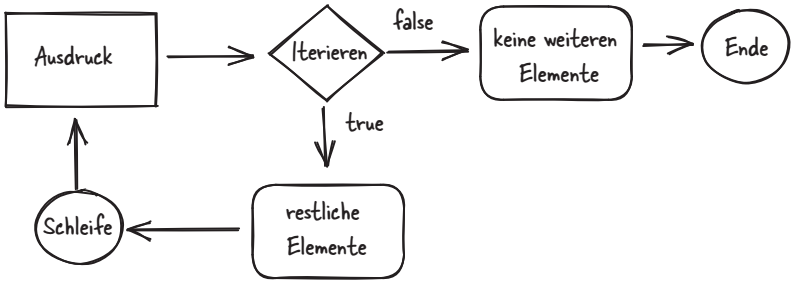

Abbildung 2-4: `for`*-Schleife*

Der typische Anwendungsfall für eine `for`-Anweisung ist das Ausführen von Iterationen für einen definierten Bereich. Sie kann auch genutzt werden, um eine `List` oder einen ähnlichen Datentyp effizient zu durchlaufen.

Aktionen abhängig von einem Wert ausführen

Problem

Sie wollen mehrere logische Prüfungen für einen Wert durchführen.

Lösung

Nutzen Sie eine `switch`-Anweisung, wenn Sie mehrere logische Anweisungen brauchen. Sind mehrere logische Prüfungen erforderlich, ist die erste Anweisung, die einem in den Sinn kommt, typischerweise eine `if`-Anweisung (die Sie in Abschnitt »Überprüfen, ob eine Bedingung erfüllt wurde« gesehen haben). Aber es kann effizienter sein, eine `switch`-Anweisung zu verwenden.

Hier ein Beispiel für eine `switch`-Anweisung:

```
void main() {
  int myValue = 2;

  switch (myValue) {
    case 1: print('Montag');
      break;
    case 2: print('Dienstag');
      break;
    default:
      print('Fehler: Wert nicht definiert?');
      break;
  }
}
```

Diskussion

Eine `switch`- (oder `case`-)Anwendung ist eine verbesserte `if`-Anweisung für das Handling mehrerer Bedingungen. In Abbildung 2-5 fungiert die `case`-Anweisung wie bei einer `if`-Anweisung, nur dass mehrere Bedingungen geprüft werden. Meist ist eine `case`-Anweisung eine lesbarere Form einer `if`-Anweisung, wenn zwei oder mehr Bedingungen auszuwerten sind.

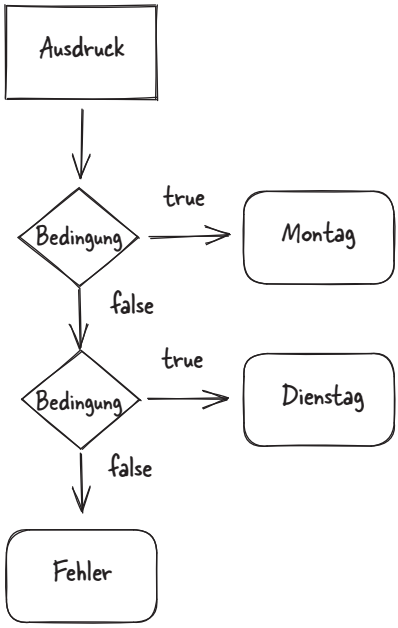

Abbildung 2-5: `switch`*-Anweisung*

Im Beispielcode besitzt die `switch`-Anweisung zwei gültige Pfade: 1 oder 2. Sie können sich sicherlich vorstellen, wie sich dieser Code auf mehr Pfade ausweiten lässt.

Die `switch`-Anweisung führt die `default`-Anweisung aus, wenn der entsprechende Wert ansonsten nicht übereingestimmt hat – es ist also eine Form von Restepfad. Das Aufnehmen solcher expliziten Anweisungen ist hilfreich, um Fehler beim Verarbeiten von Informationen zu reduzieren.

Werte mit einem Enumerator repräsentieren

Problem

Sie wollen eine Gruppe von konstanten Werten in einer Anwendung definieren.

Lösung

Nutzen Sie eine `enum` (Enumeration), um Informationen zu gruppieren, die zusammengehören.

Hier ein Beispiel für das Deklarieren und Ausgeben der Werte, die mit der `enum` assoziiert sind:

```
enum Day { sun, mon, tues }

void main() {
  print(Day.values);
}
```

Hier ein Beispiel für das Deklarieren und Ausgeben der `enum`-Referenz bei Index null:

```
enum Day { sun, mon, tues }

void main() {
  print('${Day.values[0]}');
}
```

Und ein Beispiel für den Einsatz von `values.byName`:

```
enum Day { sun, mon, tues }

void main() {
  print(Day.values.byName('mon'));
}
```

Diskussion

Die Beispiele zeigen die Nützlichkeit von `enums` beim Schreiben von Code in Dart. Aktuell unterstützt Dart nur `enum`-Definitionen auf der äußersten Scope-Ebene, Sie können die Definition also nicht in eine Klasse oder Funktion verschieben.

Ein `enum` (oder eine Enumeration) wird dazu genutzt, zusammengehörige Elemente zu definieren. Sie können sich ein `enum` als geordnete Collection vorstellen – zum Beispiel die Wochentage oder die Monate in einem Jahr. In den Beispielen lässt sich die Reihenfolge auf den Wert übertragen – der erste Monat ist beispielsweise Januar, der zwölfte Dezember.

Im ersten Beispiel sehen Sie, wie das `enum` genutzt werden kann, um eine Abfolge von Werten auszugeben. Ganz allgemein kann ein `enum` den Zugriff auf Daten ver-

einfachen. Ist ein bestimmtes Element innerhalb eines `enum` erforderlich, lässt sich das ebenfalls erreichen, wie Sie im zweiten Beispiel sehen.

Im dritten Beispiel spreche ich einen `enum`-Wert nicht über einen numerischen Index an, sondern über die Methode `byName`. Damit können Sie den Namen nutzen, der mit dem `enum`-Wert verbunden ist, um sich den Zugriff zu erleichtern. Wird die `print`-Anweisung ausgeführt, zeigt die Debugausgabe die Werte, die mit dem `enum` verbunden sind, zum Beispiel »mon«. `enum` ist weiterhin indexiert, Sie haben so aber eine bequemere Methode zum Ansprechen jedes Elements als den Zugriff über die Position als numerischen Wert.

Exception-Handling implementieren

Problem

Sie brauchen eine Möglichkeit, in einer Anwendung mit Fehlern umzugehen.

Lösung

Nutzen Sie die `try`-, `catch`- und `finally`-Blöcke, um ein Exception Management in Dart umzusetzen.

Hier ein Beispiel für das Handhaben von Exceptions in Dart:

```
void main(){
  String name = "Dart";

  try{
    print ('Name: $name');
    // Die folgende Zeile führt zu einem RangeError
    name.indexOf(name[0], name.length - (name.length+2));
  } on RangeError catch (exception) {
    print ('Exception: $exception');
  }
  catch (exception) {
    print ('Catch Exception: $exception');
  } finally {
    print ('Abgeschlossen!');
  }
}
```

Diskussion

Der Beispielcode definiert die relevanten Abschnitte und erzeugt einen `String` mit dem Wort »Dart«.

In Abbildung 2-6 allokiert eine `String`-Variable vier Speicherstellen. Um eine Exception zu erzeugen, wird die `indexOf`-Methode mit einem ungültigen Bereich genutzt (eine Stelle mehr, als die `String`-Variable lang ist). Greift man über die

letzte Stelle im vom `String` allokierten Speicher zu, führt das zu einer `RangeError`-Exception. Die Exception gibt explizit an, welchen Fehler sie sieht – im Beispiel ist der für den Index angegebene Bereich für die deklarierte Variable ungültig.

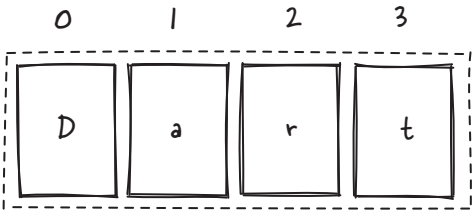

Abbildung 2-6: Beispiel für Exception-Handling

Ein `try`-Block wird für das normale Verarbeiten des Codes genutzt. Dieser Codeblock wird ausgeführt, bis ein Ereignis darauf hinweist, dass etwas Ungewöhnliches passiert. Typischerweise werden Sie Ihren Code in einem `try`-Block verpacken, um Exceptions abzufangen.

Ein `on`-Block dient dazu, mit einer bestimmten Art von Exception umzugehen, die ausgelöst wurde. Im Beispiel wird der `RangeError` berücksichtigt. Wollen Sie eine bestimmte Art von Exception verarbeiten, nutzen Sie in Ihrem Exception-Block das Schlüsselwort `on`.

Ein `catch`-Block wird genutzt, um allgemein auf ungewöhnliche Ereignisse reagieren zu können. Mit einem `catch`-Block haben Sie die Möglichkeit, sicher zu einem normalen Verhalten zurückzukehren oder sich mit dem eingetretenen Ereignis zu befassen. In den meisten Situationen werden Sie die Art von Exception nicht identifizieren können, die beim Ereignis ausgelöst wurde. Nutzen Sie den `catch`-Block, damit Ihr Code vernünftig reagieren kann, wenn ein Fehler eintritt.

Ein `finally`-Block dient dazu, eine Aktion immer auszuführen – unabhängig davon, ob der Code erfolgreich durchlaufen oder eine Exception ausgelöst wurde. Ein `finally`-Block wird typischerweise zum Aufräumen genutzt, zum Beispiel, um offene Dateien zu schließen. Zudem wird er eine Nachricht ausgeben, die zeigt, dass die Verarbeitung unabhängig vom Auftreten der Exception abgeschlossen wurde.

Das Exception-Management ist sicherlich eine Art von Ablaufsteuerung, wenn auch nicht im klassischen Sinne. Nimmt die Komplexität Ihrer Anwendung zu, wird es aber immer wichtiger, es zu integrieren.

KAPITEL 7

Einführung in Flutter

In diesem Kapitel starte ich eine Reise mit dem Framework Flutter und konzentriere mich auf einige seiner Grundlagen.

Für mich ist der beste Ausgangspunkt für eine Flutter-Anwendung ein Diagramm, wie meine Anwendung aussehen und arbeiten wird. Haben Sie einen Design-Hintergrund, sind Sie vielleicht mit dem Begriff *Wireframe* vertrauter. Auf jeden Fall werden Sie eine repräsentative Version des Anwendungsdesigns erstellen.

Sind Sie mit dem Erstellen von Benutzeroberflächen in Flutter erst einmal vertrauter, können Sie Inspiration auf Seiten wie *https://dribble.com* finden. Ich stöbere gerne auf solchen Seiten, um mir zu überlegen, wie ein Design aussehen könnte, und dann die Anforderungen in einer Folge von Bildern (zum Beispiel Wireframes) zu beschreiben.

Das Flutter-Team unterstützt Sie darin, indem es eine Vielzahl von Templates bereitstellt, mit denen Sie in die Programmierung Ihrer Anwendung einsteigen können. Haben Sie beim Code eine Grundlage, müssen Sie sich mit den Unterschieden zwischen zustandsbehafteten und zustandslosen Widgets vertraut machen, was beim Erstellen Ihrer Designs eine immer wiederkehrende Frage ist. Dankenswerterweise bedeutet Flutters Fähigkeit, sehr komplexe Oberflächen zu erstellen, dass sich diese Investition wirklich auszahlt.

Wenn es um Flutter geht, müssen Sie vor allem Widgets verstehen und begreifen, wie man sie nutzt, um Komponenten auf dem Bildschirm darzustellen. Ich beginne mit dem Mocken einer Oberfläche und dem Erstellen eines Programmierrahmens für eine Flutter-Anwendung, bevor ich definiere, was ein Widget ist. Dann zeige ich, wie Widgets genutzt werden, um eine Benutzeroberfläche zusammenzustellen. Schließlich gehe ich noch kurz auf die Struktur des Widget-Baums ein, damit Sie besser verstehen, wie sie in Flutter zum Einsatz kommt, um die Interaktion zwischen Komponenten darzustellen.

Eine Anwendungsoberfläche mocken

Problem

Sie wollen eine Oberfläche mocken, um das Layout zu verstehen, bevor Sie eine Flutter-Anwendung erstellen.

Lösung

Nutzen Sie ein Grafikpaket, um Ihre Anwendung zu designen. Das Erstellen eines Wireframes Ihrer Applikation kann Ihnen dabei helfen, besser zu verstehen, wie sie funktioniert.

Hier ein paar Beispiele für Pakete, die dabei helfen können – abhängig von Ihrem Budget und Ihrem Einsatzzweck:

Produkt	Link	Preis	Beschreibung
Excalidraw	*https://excalidraw.com*	kostenlos	Ein webbasiertes, allgemein nutzbares Tool zum Erstellen von Grafiken
Figma	*https://figma.com*	kostenlos/kostenpflichtig	Eine Lösung zum Designen und Erstellen von generierten UI-Templates ohne Code
FlutterFlow	*https://flutterflow.io*	kostenlos/kostenpflichtig	Interaktive UI-Templates und -Komponenten, die Flutter-Code erzeugen

Diskussion

Das Mocken einer Oberfläche ist eine ausgezeichnete Möglichkeit, mit einem visuellen Framework wie Flutter loszulegen. Es gibt viele Wege, eine Benutzeroberfläche zu designen – von freien Online-Tools bis hin zu dedizierten Anwendungen, die spezifisch für Flutter erstellt wurden.

Beim Anlegen eines Mocks für eine Anwendung versuche ich, die Oberfläche aus Sicht der zu verwendenden Widgets umzusetzen. Das macht es leichter, bestimmte Designs zu bauen. Haben Sie es mit komplexeren Designs zu tun, führt der Aufbau eines Verständnisses für die Anforderungen der Applikation zu einer saubereren Oberfläche und zu einer besseren Design-Ästhetik.

In Abbildung 7-1 sehen Sie ein Beispiel für eine Ausgabe mit Excalidraw für die erste Flutter-Anwendung, die ich erstellt habe.

Im Diagramm habe ich die Funktionalität mit aufgenommen – zwei Seiten und eine Navigationsleiste. Ich habe zudem angemerkt, wie der Wechsel zwischen den Bildschirmen funktionieren soll. Das Aufschlüsseln der Oberfläche ist eine gute Möglichkeit, zu lernen, wie die verschiedenen Widgets interagieren. Zudem hilft es beim

Finden der passenden Lösung, die korrekten Begriffe für Widgets und so weiter zu kennen. Diese Anwendung ist hier zwar nicht sehr kompliziert, aber sie hat mir dabei geholfen, die Grundlagen des Aufbaus mit Widgets unter Flutter zu erlernen.

Diese Art von Oberflächen ist in Flutter-Anwendungen sehr verbreitet. Es ist entscheindend, Widgets wie ListView, Text oder Image kennenzulernen, aber auch den Umgang mit Gesten und der Navigation.

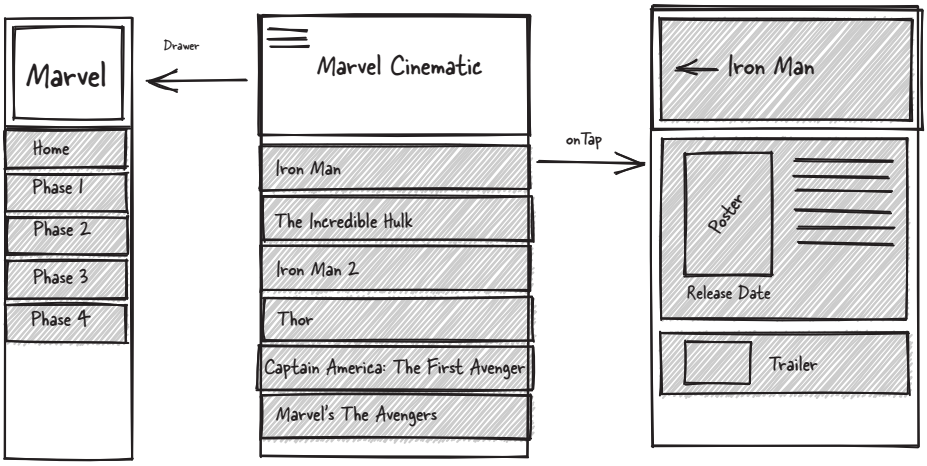

Abbildung 7-1: Wireframe-Mockup-Design

Eine Projektgrundlage in Flutter erstellen

Problem

Sie wollen eine neue Flutter-Anwendung erstellen, die auf einem Template basiert.

Lösung

Nutzen Sie ein Flutter-Template, um mit Ihrer Anwendung zu beginnen. Sie müssen nicht alles von Grund auf selbst erstellen, weil Flutter eine ganze Reihe von Anwendungs-Templates bereitstellt. Es gibt verschiedene Templates, die die grundlegenden Einstellungen mitbringen.

In aktuelleren Versionen des Flutter-Frameworks wurde an umfangreichen Beispielen gearbeitet. Nutzen Sie diese Codevorlagen, um Ihre Fähigkeiten zu verbessern und übliche Anwendungs-Patterns zu verstehen, die von Expertinnen und Experten entwickelt wurden.

Hier ein paar Templates, die mit dem Flutter-Framework mitkommen, und mit denen Sie einsteigen können:

Typ	Beschreibung
app	Das ist der Standard für `flutter create` und es wird genutzt, um eine Flutter-Anwendung zu generieren.
module	Diese Option ermöglicht es Ihnen, ein Modul zu erstellen, das mit anderen Anwendungen zusammenarbeiten kann.
package	Diese Option ermöglicht ein gemeinsam nutzbares Flutter-Projekt.
plugin	Diese Option stellt eine API-Basis für den Einsatz mit Android und iOS zur Verfügung.
skeleton	Diese Option stellt eine Best-Practices-Anwendung bereit, die auf einer Detail View basiert.

Indem Sie die Template-Anweisung anfügen (also `--template` oder `-t`), können Sie Flutter mitteilen, dass beim Erstellen ein Template zu nutzen ist. Dies sind ein paar Beispiele für den Einsatz von Templates.

Eine Standardanwendung erstellen:

```
flutter create my_awesome_app
```

Ein Modul erstellen:

```
flutter create -t module my_awesome_module
```

Ein Paket erstellen:

```
flutter create -t package my_awesome_package
```

Ein Plugin erstellen:

```
flutter create -t plugin my_awesome_plugin --platforms=web,android
```

Beim Erstellen eines Plugins müssen Sie die zu unterstützenden Plattformen angeben.

Um einen Anwendungsrahmen zu schaffen:

```
flutter create -t skeleton my_awesome_skeleton
```

Diskussion

In den gezeigten Beispielen sehen Sie, dass das Flutter-Tool viel grundlegenden Code erzeugt, mit dem Sie loslegen können. Bei dieser Aufgabe spielt der Befehl `flutter create` eine zentrale Rolle. Er ermöglicht es Ihnen, anzugeben, was für Code erzeugt werden soll, zum Beispiel Module, Pakete oder Plugins.

`flutter create` besitzt zudem eine Option, mit der Code offline erzeugt werden kann. Dazu geben Sie einfach `flutter create --offline [action]` ein. Ich finde diese Option ausgesprochen praktisch, wenn ich in einer Umgebung arbeite, die keine gute Internetverbindung besitzt. Ihre Erfahrung kann anders aussehen, weil dafür der Pub Cache verfügbar sein muss, und ich habe gelegentlich auch schon Abbrüche bei der Verwendung in Android Studio erlebt.

Erstellen Sie ein Projekt, das auf einem Template basiert, müssen Sie das Device beachten, das aktuell auf Ihrem Rechner verfügbar ist. Neben Templates können Sie auch auf Beispielcode aus der API-Dokumentation (*https://docs.flutter.dev*) zurückgreifen. Um Code von dieser Site zu verwenden, müssen Sie die Beispiel-ID referenzieren, die sich auf der Seite für das zu nutzende Widget befindet. Im folgenden Beispiel finden Sie den Code auf der Webseite zu GestureDetector (*https://oreil.ly/_0SjI*):

```
flutter create -s widgets.GestureDetector.1 my_awesome_sample
```

Mit den Beispielen können Sie schnell auf die vielen Inhalte zurückgreifen, die online zur Verfügung stehen. Als Entwickler finde ich persönlich es vorteilhaft, neben der gewünschten Host-Plattform auch das Web als Ziel mit einzubeziehen. Das ist sinnvoll, weil Sie die Anwendung so sehr viel besser in einem Browser testen können und sich auch inkrementelle Tests dort besser und einfacher umsetzen lassen. Während der Entwicklungsphase kommt man so mit kleinen und großen Verbesserungen schneller voran.

Das Debug-Banner von Flutter entfernen

Problem

Sie suchen einen Weg, das Debug-Banner aus Ihrer Flutter-Anwendung zu entfernen.

Lösung

Nutzen Sie `debugShowCheckModeBanner`, um das Debug-Banner zu entfernen, das für Ihre Flutter-Anwendungen genutzt wird.

Hier eine Beispiel-Flutter-Anwendung, die zeigt, wie Sie die Debug-Eigenschaft deaktivieren:

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    const title = 'Debug-Beispiel';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const Text("Debug-Banner entfernt"),
      ),
      debugShowMaterialGrid: false,
      debugShowCheckedModeBanner: false,
    );
  }
}
```

Diskussion

Der Beispielcode zeigt, wie Sie das Debug-Flag aus Ihrer Anwendung entfernen.

`debugShowCheckModeBanner` erwartet einen booleschen Wert, der besagt, ob die Information angezeigt werden sollte. Im Beispiel wird die »Debug«-Nachricht abgeschaltet, indem die Eigenschaft auf `false` gesetzt wird.

Flutter nutzt hier als Standardwert `true`. Beim Entwickeln muss man diesen Wert explizit auf `false` setzen, um das temporäre Banner aus der Anwendung zu entfernen. Die folgende Tabelle führt die verschiedenen Einstellungen auf, mit denen der Debug/Release-Anwendungsstatus gewählt werden kann:

Modus	Eigenschaft	Anmerkung
Debug	debugShowCheckedModeBanner	Das Banner lässt sich über den booleschen Wert steuern. `true` gibt es aus – der Standard für neue Anwendungen. Setzt man die Eigenschaft auf `false`, wird das Banner aus Ihrer Anwendung entfernt.
Release	debugShowCheckedModeBanner	Das Banner wird im Release-Modus nicht angezeigt – unabhängig vom gewählten Wert.

Die Einstellung `debugShowMaterialGrid` zeigt ein Gitter in Ihrer Anwendung an. Haben Sie Probleme, etwas auf dem Bildschirm an der richtigen Stelle auszugeben, kann diese Einstellung nützlich sein, wenn Sie sie temporär in Ihrer Applikation aktivieren. Der Standardwert für diese Option ist `false` – Sie müssen sie also nur bei Bedarf aktivieren. Um diese Einstellung nutzen zu können, muss sich Ihre Anwendung im Debug-Modus befinden.

Widgets verstehen

Problem

Sie wollen verstehen, wie Flutter Widgets einsetzt, um über mehrere Plattformen hinweg eine konsistente Darstellung zu erreichen.

Lösung

Widgets sind Komponenten, die Elemente auf dem Bildschirm repräsentieren – zum Beispiel Text, Bilder oder Listen. Beim Einstieg in Flutter kann es schwierig sein, zu verstehen, dass ein Großteil des Codes ein Widget repräsentiert.

Diskussion

Das Tolle an Flutter-Widgets ist, dass man sie kombinieren kann – Sie können also die Funktionalität verbessern, um neue Widgets zu erstellen. Das Aufbauen auf bestehenden Widgets macht das Ganze ausgesprochen mächtig, da Sie nicht alles neu erfinden müssen, sondern bestehende Komponenten direkt verwenden können.

Sie müssen dabei immer im Hinterkopf behalten, dass die Widget-Klasse eine unveränderbare Beschreibung der Benutzeroberfläche ist und in die Elemente umgewandelt werden kann, die mit dem Render-Baum verbunden sind.

Beim Erstellen einer Benutzeroberfläche werden Sie die meiste Zeit Widgets verwenden, kombinieren und neu erstellen, um die geforderte Funktionalität zu erhalten. Der interessante Teil dieser Arbeit ist das Kombinieren bestehender Widgets, um ein gewünschtes Ergebnis zu erhalten.

Den Widget-Baum verstehen

Problem

Sie wollen verstehen, wie Flutter einen Widget-Baum nutzt, um die Benutzeroberfläche aufzubauen.

Lösung

Widgets werden von Flutter genutzt, um eine View in einer Anwendung zusammenzustellen. Zum Erstellen leistungsfähiger Benutzeroberflächen ist es notwendig, Widgets zu kombinieren, und die Verbindung zwischen den Widgets wird über eine Baumstruktur erreicht. Die zu erstellende View hängt von den verwendeten Widgets und der Reihenfolge ab.

Diskussion

Wie der Name schon sagt, schafft ein Widget-Baum Beziehungen zwischen den genutzten Komponenten. Eine typische Flutter-Anwendung basiert auf einer Reihe von Widgets, wie in Abbildung 7-2 dargestellt ist.

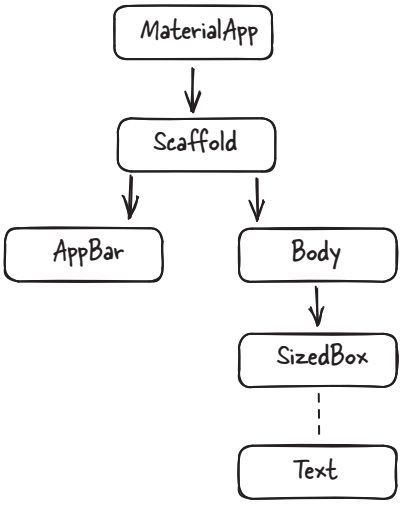

Abbildung 7-2: Widget-Baum

Aus dieser Abbildung sehen Sie, dass die `MaterialApp` das übergeordnete Element für die Anwendung ist. Dort kommt ein `Scaffold`-Widget zum Einsatz, um in untergeordnete Elemente zu verzweigen, die verschiedene Funktionalität anbieten.

Einer der wichtigsten Aspekte bei der Entwicklung mit Flutter ist das Refaktorieren Ihres Codes, sodass dieser dem Widget-Baum Ihrer Anwendung gerecht wird. Beim Einstieg in Flutter sieht man häufig sehr große Widget-Bäume, die dem in Abbildung 7-2 ähneln. Häufig trennt man dann den Build-Kontext (und spezifisch die Rendering-Schleife) ab, um die Gesamt-Rendering-Performance zu verbessern. Einen Einstieg in das Refaktorieren finden Sie in Abschnitt »Die Scaffold-Klasse einsetzen«.

Die Rendering-Performance von Widgets verbessern

Problem

Sie wollen verstehen, wie Flutter Widgets rendert, um performantere Anwendungen zu bauen.

Lösung

Flutter-Widgets sind Teil einer Rendering-Schleife, die dazu dient, die diversen Komponenten auf dem Bildschirm anzuzeigen. Da ein Widget die übergeordnete Komponente anderer Widgets sein kann, ist es manchmal erforderlich, aus Performance-Gründen einen Build-Kontext zu erstellen. Dieser delegiert das Management der zugehörigen Widgets sehr effektiv.

Diskussion

Erstellen Sie eine sehr komplexe Hierarchie, kann diese aus Sicht der Anwendungs-Performance zu einer sehr aufwendig zu wartenden Beziehung werden. Immer dann, wenn Ihre Anwendung dazu aufgefordert wird, Informationen zu rendern, wird sie versuchen, zu ermitteln, ob ihr schon die aktuellsten Informationen zur Verfügung stehen oder ob ein Aktualisieren erforderlich ist.

In Abbildung 7-3 beobachtet die Render-Schleife jede Statusänderung, die mit einem Widget verbunden ist. Wird eine Aktualisierung gefunden, führt das zu einem erneuten Bauen des zugehörigen Widget-Baums für den Build-Kontext.

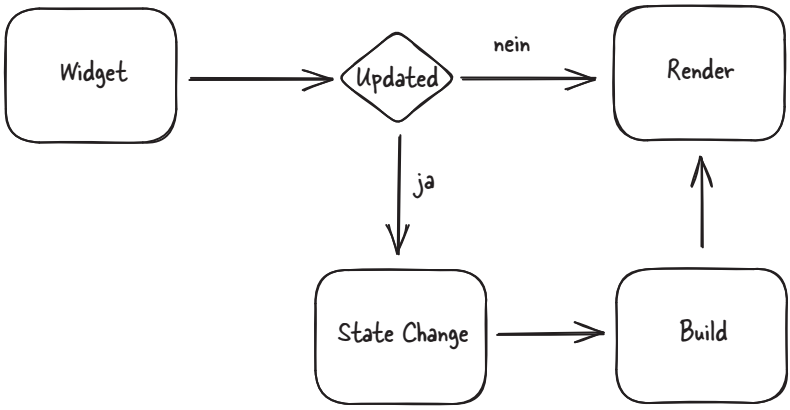

Abbildung 7-3: Rendering-Schleife von Flutter

Besitzt Ihr Widget-Baum eine große Hierarchie, wird diese neu gebaut werden müssen, was die Gesamt-Performance der Anwendung beeinflussen kann. Insbesondere wenn Ihr Build-Kontext einen Status enthält, ist das einer der häufigsten Gründe für Peformance-Flaschenhälse bei der Flutter-Entwicklung. Das Isolieren des Status in einem kleinen Build-Kontext wird die Gesamt-Performance deutlich verbessern, da sich die Statusänderung dann nur auf das relevante Widget auswirkt und nicht auf den gesamten Widget-Baum.

KAPITEL 14

Die Flutter-Benutzeroberfläche testen

In diesem Kapitel lernen Sie, wie Sie Testfälle für die Benutzeroberfläche (User Interface, UI) für Ihren Flutter-basierten Code schreiben. Das Erstellen eines Tests für das UI kann viel Aufwand sein, weil die Anwendung, die für das Abfragen der Benutzeraktionen zuständig ist, wissen muss, welche Elemente sich auf dem Bildschirm befinden.

Es gibt eine Reihe von Vorgehensweisen, die beim Testen des UI Ihrer Anwendung zum Einsatz kommen können. Zuerst beschreibe ich automatisierte Widget-Tests. Dann zeige ich externe Tools, die die gleiche Funktionalität bieten. Haben Sie so etwas wie eine Continuous Integration Pipeline im Einsatz, ist es besonders hilfreich, Tests außerhalb der Flutter-Umgebung ausführen zu können, da dadurch weniger Platform Scaffolding erforderlich ist.

Sie werden lernen,

- wie automatisierte Widget-Tests funktionieren,
- wie Sie automatisierte Widget-Tests integrieren,
- wie der Flutter Driver zu nutzen ist und
- wie Sie mit der Firebase Testing Suite arbeiten.

Am Ende des Kapitels werden Sie die verfügbaren Optionen kennen und dazu in der Lage sein, diese Techniken in Ihrem eigenen Projekt zu verwenden. Es sei darauf hingewiesen, dass das automatisierte Testen eines UI ein sich weiterhin entwickelndes Thema ist, daher kann es bei Ihnen schon wieder anders aussehen.

Automatisierte Widget-Tests in Flutter

Problem

Sie suchen einen Weg, ausgefeilte UI-Tests durchzuführen, die auch Benutzerinteraktion simulieren.

Lösung

Nutzen Sie Widget-Tests, um Ihrer Anwendung noch mehr vertrauen zu können. Insbesondere können Sie Tests für Widget-Elemente wie FloatingActionButtons, Text und ListViews aufrufen. Nehmen Sie diese Funktionalität in Ihre Anwendung mit auf, erhalten Sie zusätzliche Abdeckung, um noch weniger Fehler zu übersehen. In Abbildung 14-1 dient die Flutter-Counter-App dazu, die Methoden des automatisierten Testens zu illustrieren.

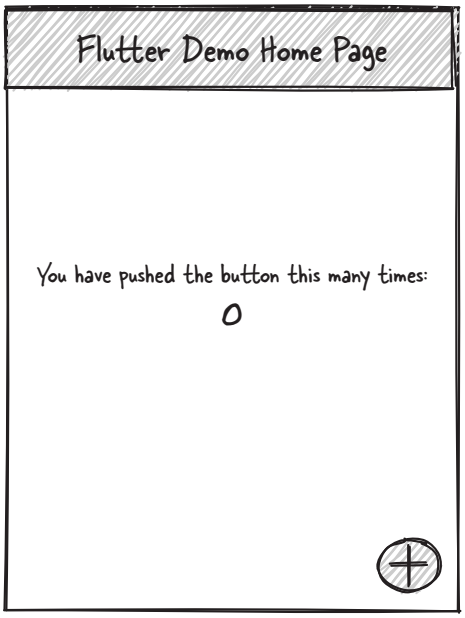

Abbildung 14-1: Ein Flutter-UI testen

Um die Counter-App zu testen, brauchen Sie ein bisschen Code. Der folgende Testcode dient dazu, die Elemente auf dem Bildschirm zu überprüfen, die mit der Anwendung verbunden sind:

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';

import 'package:test_widget_app/main.dart';

void main() {
  testWidgets('Counter Smoke Test', (WidgetTester tester) async {
    // Die App bauen und einen Frame auslösen.
    await tester.pumpWidget(const MyApp());

    // Prüfen, dass der Zähler bei 0 beginnt.
    expect(find.text('0'), findsOneWidget);
    expect(find.text('1'), findsNothing);
```

```
    // Auf das '+'-Icon tippen und einen Frame auslösen.
    await tester.tap(find.byIcon(Icons.add));
    await tester.pump();

    // Prüfen, dass der Zähler hochgezählt wurde.
    expect(find.text('0'), findsNothing);
    expect(find.text('1'), findsOneWidget);
  });
}
```

Diskussion

Der Beispielcode, der zur Flutter-Projektstruktur hinzugefügt wurde, nutzt eine Flutter-Demo-App als Beispiel. Beim Erstellen dieses Templates, das vom Flutter-Framework generiert wurde, handelt es sich bei den Tests um Widget-Tests.

Widget-Tests haben eine ähnliche Signatur wie Unit-Tests. Der Unterschied ist der Einsatz von `testWidgets` und `WidgetTester`, um eine Anwendung zu überprüfen. Das Hinzufügen von Widget-Tests führt zu einer umfassenderen Abdeckung gegenüber Unit-Tests. Im Beispiel ist das UI mit dem Test verbunden, um die Benutzeraktionen abdecken zu können. Das führt zwar zu einem breiteren Funktionsspektrum, erhöht aber auch den Aufwand, der mit dem Bauen von Tests verbunden ist.

Mit einem `tester.pumpWidget` wird der zu testende Bildschirm »aufgepumpt«. Haben Sie eine Anwendung mit mehreren Bildschirmen, müssen Sie darauf achten, dass diese Methode korrekt aufgerufen wird. In den meisten Beispielen wird die Klasse `MyApp` genutzt, die mit der MaterialApp verbunden ist:

```
// Die App bauen und einen Frame auslösen
await tester.pumpWidget(const MyApp());
```

Nutzen Sie einen anderen Bildschirm, der dieser Klasse untergeordnet ist, werden Sie die Deklaration wie folgt anpassen müssen:

```
// Die App bauen und einen Frame auslösen:
await tester.pumpWidget(MaterialApp(detail: DetailPage()));
```

Auf die Informationen auf dem Bildschirm können Sie auf verschiedenen Wegen zugreifen. Im Beispielcode sind die Elemente auf dem Bildschirm gut genug unterscheidbar, sodass die Methode `find.text` ausreichen wird. Ist in Ihrer Anwendung mehr los und gibt es mehr Elemente, kann eine alternative Methode notwendig sein. Zum Glück besitzt die Methode `find` eine Reihe von Alternativen, die die meisten Situationen abdecken. Ich empfehle Ihnen, die Methode `find` als Lesezeichen zu hinterlegen, wenn Sie Widget-Tests erstellen wollen. Bei mir kommen `find.byType`, `find.byWidget` und `find.text` am häufigsten zum Einsatz.

Der typische Anwendungsfall für das Testen von Widgets ist das Überprüfen der Interaktivität zwischen Bildschirm und Anwender beziehungsweise Anwenderin. Widget-Tests können zwar sehr schnell erstellt werden, aber überlegen Sie sich, ob

die Anwendungsfälle für ausreichend Abdeckung sorgen. In vielen Fällen – abhängig von der Art Ihrer Anwendung – kann eine Kombination aus Unit und Widget-Tests das beste Vorgehen sein.

Automatisierte Widget-Tests ausführen

Problem

Sie wollen testen, ob das Widget UI die Anforderungen erfüllt.

Lösung

Konfigurieren Sie Ihre Anwendung so, dass sie Schlüssel im Anwendungscode einsetzt. Dann können die Widget-Tests die relevanten Elemente auf saubere Art und Weise finden.

Im Beispiel sind Felder für den Benutzernamen und das Passwort definiert. Der Widget-Test nutzt die Methoden `pumpWidget` und `pump`, um mit dem Widget zu interagieren:

```
void main(){
  testLoginWidget("Login sollte möglich sein", (WidgetTester testWorker) async {
    // Arrange
    final testUsername = find.byKey(ValueKey("testUsername"));
    final testPassword  = find.byKey(ValueKey("testPassword"));
    final testLoginBtn  = find.byKey(ValueKey("testLoginBtn"));

   // Act
   await testWorker.pumpWidget(MaterialApp(home: Home()));
   await testWorker.enterText(testUsername, "username");
   await testWorker.enterText(testPassword, "password");
   await testWorker.tap(testLoginBtn);
   Await testWorker.pump();

   // Assert
   expect(find.text("Login credentials supplied"), findsOneWidget);
  });
}
```

Diskussion

Im Beispielcode erstelle ich einen Widget-Test, um die Homepage zu überprüfen. Sie enthält eine Reihe von Bildschirmelementen, die eine Benutzerinteraktion erfordern. Zuvor habe ich Unit-Tests erstellt, um sicherzustellen, dass die Codeabschnitte meine Anforderungen erfüllen. Das Hinzufügen eines Widget-Tests sorgt für eine zusätzliche Verbesserung Ihrer Anwendung.

Um Ihre Anwendung für Widget-Tests zu konfigurieren, ist zusätzlicher Aufwand nötig (zum Beispiel die zu testenden Bildschirminformationen ermitteln), um Ihren Code so zu organisieren, dass leichter auf die Informationen zugegriffen werden kann. Der Hauptanwendungsfall für diese Art von Tests ist ein UI, das viel wiederholtes Testen benötigt.

Integrationstests mit Flutter Driver durchführen

Problem

Sie wollen die gesamte Anwendung testen können.

Lösung

Nutzen Sie Flutter Driver, da dies ein Oberflächensteuerungs-Paket bereitstellt, mit dem automatisiert mit einer Anwendung interagiert werden kann. Die Interaktion automatisiert die normalen Aktionen, die von einer Benutzerin oder einem Benutzer der Anwendung durchgeführt würden – zum Beispiel das Eingeben von Text, das Auswählen von Elementen oder das Drücken von Buttons.

Um einen Integrationstest zu starten, erstellen Sie in Flutter eine einfache Schnittstelle:

```
import 'package:flutter/material.dart';
import 'package:flutter_driver/driver_extension.dart';

void main() {
  enableFlutterDriverExtension();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  …
}

class MyWidget extends StatelessWidget {
  …
}
```

Erzeugen Sie nun eine Integrationstest-Anwendung:

```
final txtUsername = find.byType(Text);
final btnAddition = find.byType(FloatingActionButton);

FlutterDriver driver;

setUpAll(() async {
  driver = await FlutterDriver.connect();
});
```

```
tearDownAll(()) async {
  if (driver != null) {
    driver.close();
  }
}

test ('Benutzernamen eingeben und Button drücken', ()async {
  await driver.tap(txtUsername);
  await driver.enterText("Martha Kent")
  await driver.tap(btnAddition);
  await driver.waitFor(find.text("Willkommen"));
});
```

Diskussion

Im Beispielcode wird der Flutter Driver zur Anwendung hinzugefügt, um die notwendigen Integrationstests ermöglichen zu können.

Um diese auch auszuführen, starten Sie den Prozess an der Befehlszeile wie folgt:

```
flutter drive --target=test_driver/main.dart
```

Wird der Integrationstest gestartet, versucht er, die Anwendung zu starten und dann wie ein User den Befehl auszuführen.

Der Integrationstest nutzt eine asynchrone Schnittstelle, um Ihre Anwendung zu durchlaufen. Haben Sie schon mit Selenium oder Puppeteer gearbeitet, werden Sie mit dieser Art des Testens vertraut sein. Der größte Aufwand dabei ist, die Validität der Tests zu ermitteln. Die auf dem Bildschirm sichtbaren Elemente haben meist einen zugewiesenen Timeout, der jede Interaktion abhängig von der asynchronen Wartezeit macht.

Integrationstests erfordern mehr Aufwand gegenüber Unit und Widget-Tests. Aus diesem Grund werden solche Tests in der Entwicklung gerne vermieden und andere Arten werden vorgezogen. Angesichts der Interaktion mit dem UI können Änderungen daran dazu führen, dass die Integration brüchig wird. So könnte eine Änderung am UI beispielsweise zu einem anderen logischen Ablauf führen.

In den meisten Fällen werden Sie beim Ausführen von Integrationstests eine Testumgebung nutzen (manchmal als Staging bezeichnet). Das erlaubt es Ihnen, die erstellten Daten wieder zu entfernen oder alles ganz zu löschen. Idealerweise sollte das eine Umgebung sein, in der Sie Daten völlig frei laden und auch wieder löschen können.

Der typische Anwendungsfall für Integrationstests dreht sich um sich wiederholende Aufgaben. Das Automatisieren dieses Szenarios kann sehr sinnvoll sein. Denken Sie aber daran, dass der Aufwand für das Einrichten und Betreuen der Testumgebung möglicherweise sehr groß ist. Wenn Sie das nicht abschreckt, kann das automatische Durchlaufen einer Anwendung viel Zeit einsparen.

Die Kompatibilität mit Android/iOS-Geräten testen

Problem

Sie wollen durch automatisierte Tests die Geräte-Kompatibilität sicherstellen.

Lösung

Nutzen Sie die Robo-Tests der Firebase Test Labs, um einen praxisorientierten Ansatz beim Testen zu verfolgen. Die Firebase-Lösung erfordert kein zusätzliches Setup und sie erwartet nur das Anwendungs-Binary als Eingabe.

Diskussion

Firebase Test Lab bietet eine ganze Reihe an Test-Tools für Ihre Anwendung an.

Beachten Sie, dass die Firebase-Test-Suite aktuell nur auf Android und iOS ausgerichtet ist. Haben Sie auch andere Plattformen im Auge, wie zum Beispiel das Web, Windows oder Linux, sollten Sie sich auch andere Optionen anschauen.

Wollen Sie die Kompatibilität mit mobilen Geräten verbessern, ist Firebase Test Lab vermutlich die optimale Lösung für Sie und Ihr Team. Wie bei anderen Firebase-Produkten basiert die Bezahlung auf Spark (beschränkte Tests auf fünf physischen beziehungsweise zehn virtuellen Geräten) und Blaze (Abrechnung pro Minute).

Das Firebase Test Lab baut auf mehreren Produkten auf. Jedes Produkt ist darauf ausgerichtet, die wichtigsten Phasen des Quality Assurance Lifecycle abzudecken. Anwendungen können damit getestet werden, ohne zusätzlichen Programmieraufwand zu haben, weil einfach nur die Binärdatei der Flutter-Anwendung geladen wird. Firebase Test Lab kann in einen existierenden Workflow für Android-Build-Prozesse eingebunden werden, um die Anwendungsentwicklung zu orchestrieren. Die Tests können entweder auf realen oder auf virtuellen Geräten ausgeführt werden, um ein realistisches Feedback zu erhalten.

Die Anwendungssuite übernimmt die Anwendung mit Debugcode als Eingabe. Für Android muss das Android Package Kit (APK)/Android App Bundle (AAB) bereitgestellt werden, um den Prozess zu initiieren. Das Android-Binary finden Sie im Verzeichnis *build*. Zudem müssen die virtuellen Geräte ausgewählt werden, auf denen die Anwendung deployt werden soll. Es steht dabei eine große Auswahl an Devices zur Verfügung.

Das Lab-Testing kann mit dieser Testsuite auf unterschiedlichste Art und Weise ausgeführt werden. Robo-Tests (also automatisierte Tests) lassen die Anwendung auf einer Vielzahl von Android-Devices laufen. Während dieser Testläufe wird die Ausgabe erfasst – insbesondere in Bezug auf Abstürze oder Fehler. Zudem gibt es kleinere Tests, die instrumentierte Tests beinhalten (vom Entwickler/der Entwicklerin geschrieben), um zu prüfen, ob die Anwendung ihre Anforderungen erfüllt.

Robo-Tests auf Android können genutzt werden, um in einer App automatisch zu navigieren und Log-Ausgaben zu erfassen. Das ist nützlich für eine tiefgehende Fehlersuche. Robo-Tests liefern bei jedem Testlauf viele Informationen, wie zum Beispiel Screenshots und Videos des Tests. Sind iOS-Geräte Ihre Zielplattform, führt XCTest Unit, Performance- und UI-Tests für Xcode-basierte Projekte aus.

Der typische Anwendungsfall für den Einsatz des Firebase Test Lab ist das Ermitteln der eigentlichen Ursache für Fehler in einer Anwendung. Es ist eine sehr umfassende Lösung für Tests auf Geräten, daher ist es sehr empfehlenswert, diese Tests zu nutzen – und sei es nur im beschränkten Umfang.

Inhalt